EE540 Spring 2017, Assignment2, Due 15.3.2017

Microprocessor Design: Register File and ALU

The purpose of this assignment is to design a Register File (RF) and a 16-bit Arithmetic and Logic Unit (ALU) for your microprocessor. This ALU will be used in the datapath of the microprocessor, and must support signed 2's complement arithmetic and all the arithmetic and logic instructions in the baseline architecture.

Register File (RF)

In this assignment, you will design a 16-word, 16-bit RF with one write port and two read ports. This means you are going to use 16 of 16-bit registers. One of the structures that is central to the datapath is an RF. An RF consists of a set of registers that can be read from or written to. Writing a register requires (i) an address, (ii) the data to be written and (iii) a signal that controls the write timing. Reading an RF can often be controlled by just the address but may include additional logic. The RF for this project is to include an array of flip-flops or latches, muxes and demuxes.

Register Cell

Conventionally, an RF would consist of an array of static RAM cells with read/write circuitry and a sense amplifier. Though this type of implementation yields a fast, compact design, it requires much design effort, especially for the sense amplifiers. In small RFs (few registers), the support circuitry for the SRAMs is used by a small number of registers, making the SRAM-based RF larger than some other types. An alternative approach is to use a d-flip-flop or a latch consistent with your clocking scheme and an enable signal. One such example is given on the next page. Our way of implementing an RF is with multiplexors and it is conceptually simple but routing all of the mux inputs from the array cells to the muxes will create unnecessary routing bottlenecks. You may use other architectures and technologies if you want.

Drivers and Buffers

Control signals shared by all 16 bits of a register can have large capacitive loading compared to other signals in your design. The data output buses on the RF may also have large loading associated with them. These signals are candidates for the insertion of buffers. This part is optional for you. However, give it a consideration.

Bus Strategy

You can use several methods to drive values onto the microcontroller buses. Two common ones that you can use in this class are multiple buses coming out of the RF with a mux to select the proper word. Another possibility is to have tristate drivers on the outputs of each component that can drive the bus. This will require only one bus. Note that true tristate buses require keeper latches if they have active gate inputs connected to them and are allowed to float to intermediate voltages. You are free to decide which strategy you will use.

This assignment is based on EECS427-VLSI Design I Class at the University of Michigan, given by Professor Dennis Sylvester

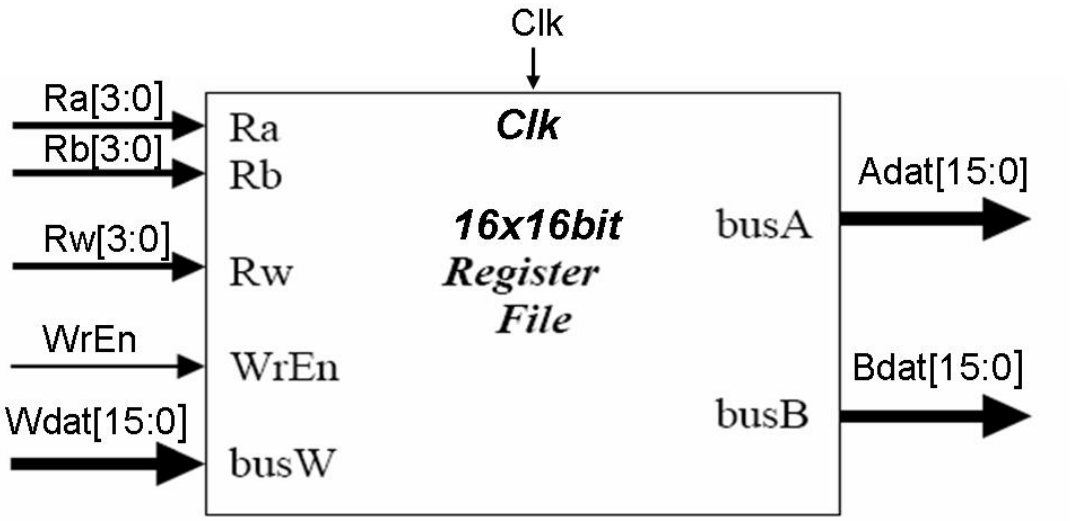


Figure 1. A register file component. Adat[15:0] gets RF[Ra], Bdat[15:0] gets RF[Rb]. Ra and Rb are 4-bit addresses that choose the desired register cell out of 16 of them. When WrEn is enabled, RF[Rw] get Wdat[15:0]. Otherwise, the registers keep their data.

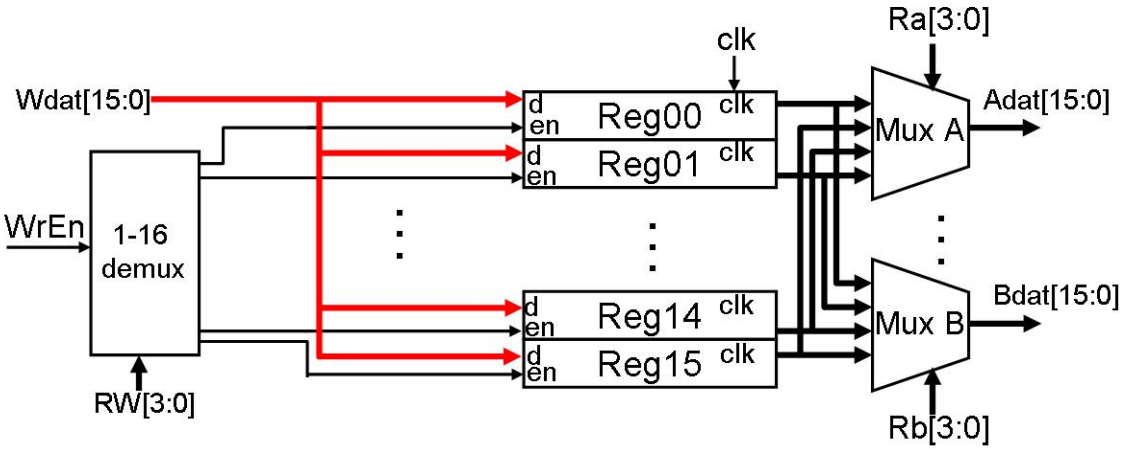


Figure 2. One example implementation of a register file.

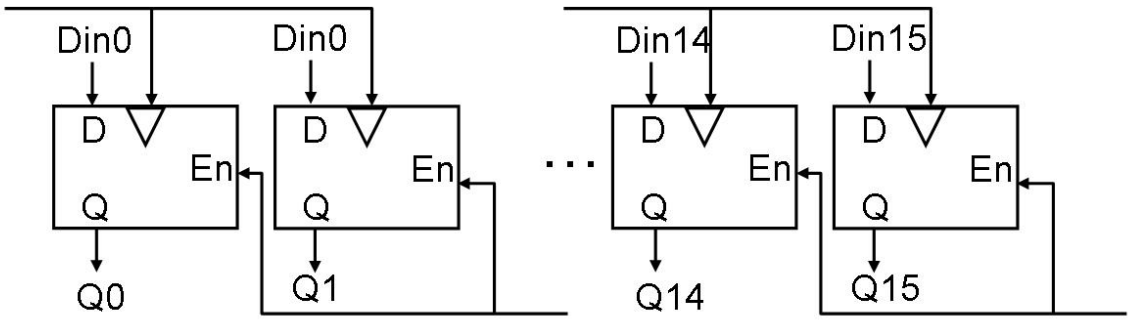


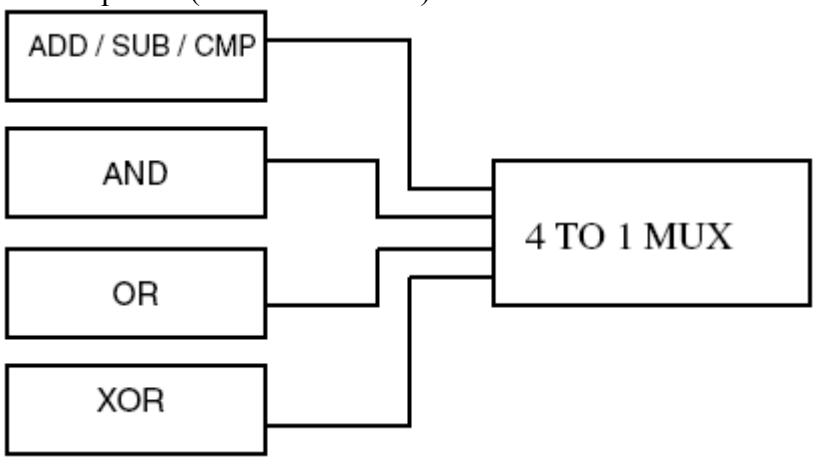
Figure 3. One example implementation of one of 16 register cell. You need to have an "enable" input of the d-flip-flops in order to be able to write to them when the write enable signal is activated. The flip flop updates its state at the clock edge only if enable is high. Otherwise it keeps its old value.

Arithmetic & Logic Unit (ALU):

The Arithmetic and Logic Unit (ALU) is the heart of your processor. The minimum set of instructions your ALU needs to support consists of (i) ADD, (ii) SUB, (iii) CMP, (iv) AND, (v) OR, (vi) XOR, and (vii) NOT. Your ALU must support both register-based operands and immediate operands. You can implement all the above instructions with the adder as the basic building block. A 2's complement subtractor and a comparator can be derived from the adder structure. 2's complement subtraction (A-B) is implemented by adding A, B_bar, and 1 (Carry_in = 1). Compare functions (GE, LE), which are used to set processor flags for conditional branches, can be implemented with a subtractor and the information from the most significant bit (sign bit). Implementing "Less Than" is a little more complicated because of overflow problems. To detect a zero output, you need a NOR tree to verify that no result bit is a 1. There are two general approaches in designing the ALU. The first one is conceptually simpler but the second one may result in a more efficient implementation.

1. Separate Logic Blocks

Your ALU must perform six different functions; this could be accomplished as shown below, with the adder unit performing three of these, and simple logic blocks performing the other three. Then a four-to-one multiplexer (or tristate buffers) could be used to select the correct function.



2. Modified Ripple Carry Adder

An alternative is to make the adder perform the logic functions as well as arithmetic functions. Consider the standard adder equations:

Sum: S = ABC + AB'C' + A'B'C + A'BC'

Carry: Cout = AB + BC + AC

If we define

Half sum: H = AB' + A'B

then we could write the adder equations as

Sum: S = HC' + H'C

Carry: Cout = AB + HC

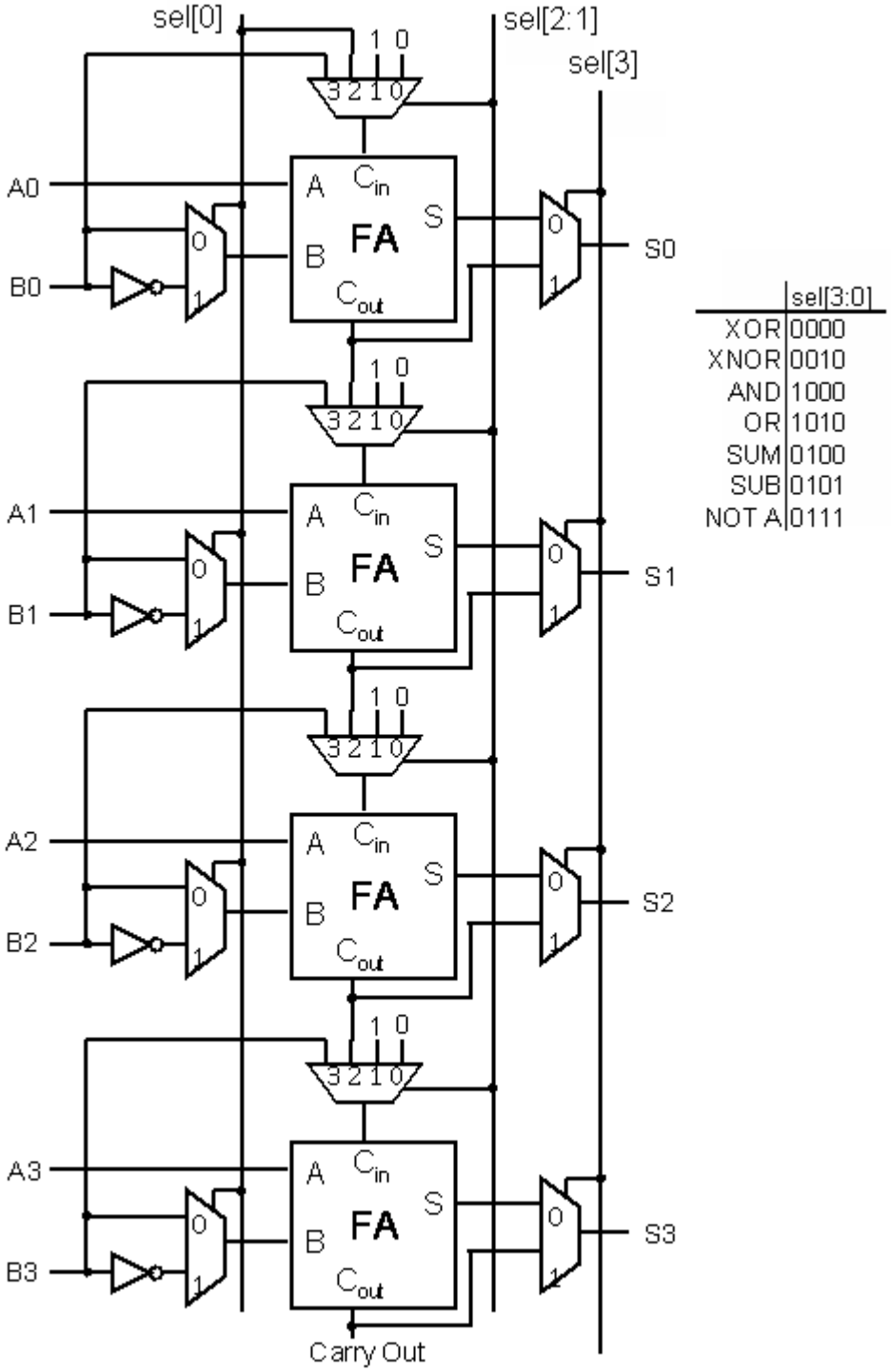
From the equations it is clear that when C is held at logical 0, the sum output is an XOR of A and B.

S = AB' + A'B (XOR operation)

Also, when C is held at logical 0, then,

Cout = AB (AND operation)

When C is logic 1, then
Cout = AB + A + B = A + B (OR operation)
This shows that appropriate switching of the carry line between adder elements will give the ALU logical functions. An example block diagram is shown below.



3. Adder Designs

Any of a variety of 16-bit adder designs could be used in your processors. Whichever design you choose to implement, you would first build a 1-bit cell. Note that any number of such elements may be cascaded to form an adder of desired width. The truth table of a 1-bit adder is as follows:

C	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Hence, we may write the standard adder equation in the form
Sum S = ABC + AB'C' + A'B'C + A'BC'
Carry Cout = AB + BC + AC = AB + C(A+B)

An alternate adder structure makes use of the Propagate (P) and Generate (G) intermediate values for computation of the Sum and Carry generated for each bit.
Propagate P = A^B
Generate G = AB

Propagate bit serves as a signal that enables propagation of an input carry to output for the corresponding bit. Generate bit generates a carry at the corresponding bit regardless of the input carry. Using these two signals that can be immediately evaluated as soon as A and B bits are available in conjunction with the input carry, one can compute the Sum and Carry as:

Sum S = P^C
Carry Cout = G + PC

PSR Description

The Processor Status Register is a 16 bit scannable register that holds the processor flags, which represent the information pertaining to the current state of the microprocessor. The assignments of bits in the PSR are shown below.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	**R**	**R**	**R**	**I**	**P**	**E**	**0**	**N**	**Z**	**F**	**0**	**0**	**L**	**T**	**C**

Bits 12 to 15 are "Reserved" which means that these bits should not be written or read. Bits 3, 4 and 8 will be hardwired to zero. The bits **N**egative, **L**ow, **Z**ero, **C**arry and Over**F**low are set by the various arithmetic instructions. Unless you are implementing maskable interrupts, you can set the **E** and **I** bits to zero. Similarly, the **T**race and **P** bits can be set to zero unless you implement tracing. See the notes at the end of the handout on the instruction set for more information about these flags. Adding or subtracting two n-bit numbers can require an n+1 bit number to fully express the result. When the result requires more bits than are available, we have an overflow or carry condition. Overflow occurs for 2's complement numbers under the conditions indicated in Table 1. An easy way to determine overflow is to exclusive-OR the carry-in of the high-order bit

with the carry-out of the high-order bit. It is left as an exercise for the student to verify that this detects overflow. Carry occurs when the result of adding unsigned n-bit numbers exceeds 2^n-1, or the result of subtracting unsigned numbers is negative. The only way to determine Zero is to check that all of the bits are zero with an OR or OR-like function. The Negative bit can be set from the high-order bit. The L bit is set assuming unsigned numbers, i.e., that all 16 bits of the operand indicate values, rather than sign, or, that all numbers are positive. The compare instruction does a subtraction of the value in *src1* register from that in the *src2* register. It is not possible to get an overflow when numbers of the same sign are subtracted. In the compare instruction, operands are treated as both signed and unsigned integers at the same time; the

Table 2: Overflow Conditions.

Operation	Operand A	Operand B	Result
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

processor does not know which they are. The programmer does know what type of data the operands represent, so can choose the correct condition on which to conditionally branch or jump. Be sure to clear the flags when they should be cleared (as well as setting them at the appropriate times). Do not change flags when they should not be changed. This can be implemented by clocking the flip-flop for a particular flag bit whenever an instruction which is to set or clear the bit has been executed. You could include instructions for loading and storing the processor status register (LPR and SPR). These instructions would move data from one of the general-purpose registers into the status register, or read the status register into a general-purpose register. These instructions are not included in the baseline machine; the baseline machine requires the PSR to be modified only by arithmetic instructions and reset. If you are not implementing the LPR and SPR instructions, you can save area by just implementing the flag flip-flops which you need (5 for the baseline processor).

Implementation of the Flags

The flags for addition are straightforward. Let c_n be the carry out of the most significant bit and c_{n-1} be the carry into the most significant position. Then $C=c_n$, and $F=c_n \oplus c_{n-1}$. The most common way of doing subtraction when 2's complement numbers are used is to add the one's complement of the number to be subtracted and make the carry in to the least significant bit equal to one. In this case the carry flag is given by $C= \overline{c_n}$, and F is the same as for addition. In the compare operation (CMP, CMPI) $L= \overline{c_n}$, and N can be derived in several ways:

$$N = a_{n-1} \oplus b_{n-1} \oplus \overline{c_n} = a_{n-1} \oplus \overline{b_{n-1}} \oplus c_n = s_{n-1} \oplus c_n \oplus c_{n-1}$$

where n-1 refers to the most significant bits in computing A-B, and s_{n-1} is the sum bit.

Assignment:

Register File (RF)

- Determine your read/write methodology for your RF and select (from assignment1) a register or latch, which will serve as the basis of your register file array cell. Also determine your bussing strategy. You may want to design a new register cell. Then replicate your 16-bit register 16 times to form the array. Write Verilog codes to do all these, to implement the 16 Word 16 bit RF.
- Run ISim on the RF and verify that it functions as expected by running a few test sequences. For example, write to each location then read from each location.
- Use Xilinx to synthesize the RF based on Xilinx Spartan 6 family.
- Simulate the synthesized design in ISim to be sure it functions correctly. After the Verilog model is synthesized, it is always a good idea to functionally simulate the resulting design to ensure it functions correctly and the output matches the initial description. There are some legal, synthesizable Verilog constructs that, when synthesized, will not produce the same output as the functional description and synthesis tools have been know on rare occasions to produce incorrect results. To do this, choose the output tab and set the *Format* item to be Verilog in Xilinx ISE. Make sure that you change the filename to *some_other_name.v* or Xilinx ISE will overwrite your Verilog source file. After you have confirmed that you have changed the name in the *Filename* item, click Write.

Arithmetic & Logic Unit (ALU)

- Write Verilog codes for the 16-bit ALU and the logic to set processor flags (see PSR below).
- Use ISim to verify each of the seven functions. Test your processor flags as well. ISim traces should show that the 16-bit ALU works for all seven (more if you implement more) functions with a few tests per function.
- Use Xilinx ISE to synthesize the ALU based on Xilinx Spartan 6 family.
- Simulate the synthesized design in ISim again to be sure it functions correctly.

Deliverables

Your reports (*.doc or *.pdf) should include the snapshots for all simulation results, a summary of the synthesis results (do not simply copy/paste the *.syr file, present the results in tabular form if possible), schematics of the synthesized circuits, and your comments. Important considerations that went into the design, and extra features, if any, should also be documented. Your reports should not contain your Verilog codes, which will be submitted as individual files. Before emailing your homework, zip all relevant Verilog files including testbenches, tcl files, synthesis output files (*.syr only), and your report under a folder named as: *hw1_yourlastname_yourfirstname*. Only the following file types should be in the unzipped folder: *.doc (or *.docx or *.pdf), *.v, *.syr, *.tcl.